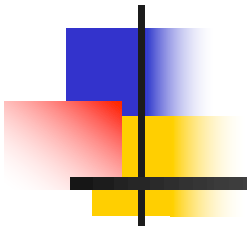


IN3030/IN4330 Uke 15, v2020

Eric Jul
PT
Inst. for informatikk

Resten av IN3030/IN4330 – v2020

- Denne forelesningen (uke15)
  - Mer om hvordan parallellisere ulike problemer
- 29. mars (uke16) – ingen forelesning – jobbar m Oblig5
- 6. mai (uke17) – ingen forelesning
- 13. mai (uke18) – review av kurs, perspektiv, litt om eksamen
- 20. mai (uke19) – ingen forelesning
- 27. mai (uke20) – meget om eksamen, evaluering
- 3.-10. juni – hjemmeeksamen via Inspera
- 10. juni kl 09:00 – frist for innlevering av eksamen i Inspera

Hva så vi på i Uke14

Oblig5

Hva skal vi se på i Uke15

I) Om program, samtidige kall og synlighet av data

- Når en tråd (main eller en av de andre) aksesserer data, hvilke er det.
- Kan de parallelle trådene og main kalle samme metode samtidig. Kan de kalle metoder fra en sekvensiell løsning? Hva skjer da ?
- Hvor mange stack-er (stabler) har vi; hva er det, og hvilke har vi?

II) Mer om to store programmer, og hvordan nå den siste av dem kan parallelliseres.

- Delaunay triangulering – de beste trekantene!
  - Brukes ved kartlegging, oljeleting, bølgekraftverk,..
  - Spill-grafikk: ved å gi tekstur, farge og glatte overflater på gjenstander, personer, våpen osv.
  - Er egentlig flere algoritmer etter hverandre. Skissering av hvordan disse kan parallelliseres.

III) Parallellisering av Oblig 5 - den konvekse innhyllinga (DKI)

- Hva er forventet O() – kompleksitet av DKI
- To strategier for parallellisering - roadmap
- Idag : rekursiv parallellisering
- Hva kan ventes av speedup på oblig 5

II) Om program, samtidige kall og synlighet av data

- Når en tråd (main eller en av de andre) aksesserer data, hvilke er det?
- Kan vi stoppe en annen tråd ?
  - Hvis ikke vi, hvem da?
- Hvor mange stack-er har vi, hva er det og hvilke har vi?
- Kan flere tråder kalle samme metode samtidig.
  - Hva skjer da ?

Når en tråd (main eller en av de andre) aksesserer (lesere/skriver) data, hvilke data er det?

- Svar: Det vanlige skopet (utsynet til deklarasjoner):
  - først lokale variable og i metoden og parametre
  - så variable og metoder i klassen man er inne i
  - Evt. så i en eller flere ytre klasser
- Dette gjelder alltid uansett hvilken tråd som kaller metoden
  - Kallstedet har sitt skop
  - Utførelsesstedet har sitt skop.

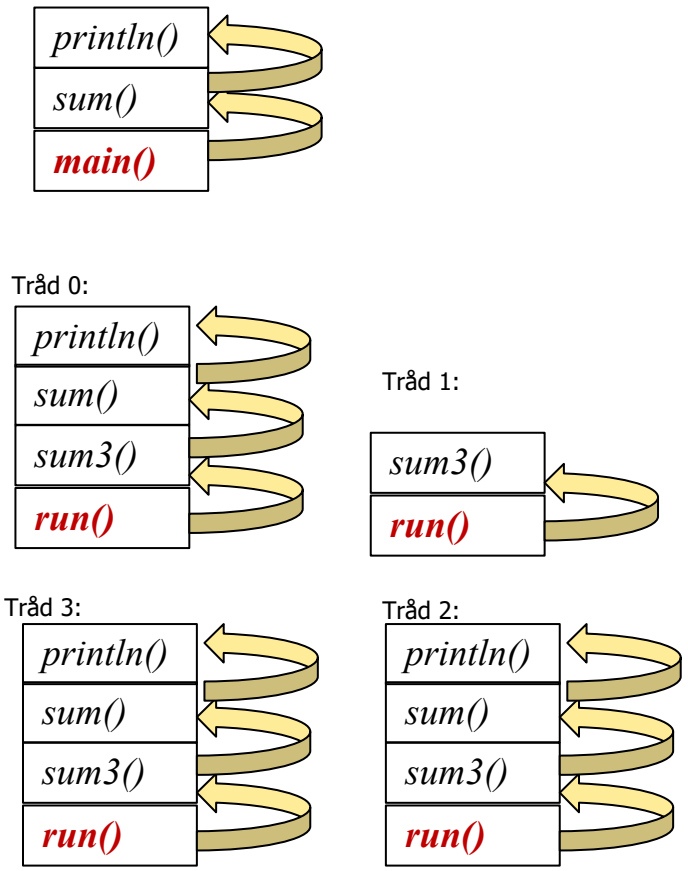
```
import java.util.concurrent.*;
class Problem {
    int [] a = new int[100]; // fyll med verdier
    long s;
     public static void main(String [] args) {
          Problem p = new Problem();
          p.utfoer(12);
     }
     void utfoer (int antT) {
          Thread [] t = new Thread [antT];
          for (int i =0; i< antT; i++)
              ( t[i] = new Thread(new Arbeider(i))).start();
           long s = sum(a);
           long t = t[0].sum2( t[0].b);

           for (int i =0; i< antT; i++) t[i].join();
     }
     long sum (int a[]) { s=0 ;
         for (int i = 0; i<a.length;i++) s += a[i];
         return s;
     }

     class Arbeider implements Runnable {
              int ind;
              int [] b= new int[200]; // fyll b[]
              long sum2 (int a[]) {long s=0 ;
                        for (int i = 0; i<a.length;i++) s += a[i];
                  return s;
              }
               long sum3(int [] c) { return sum(c);}

              Arbeider (int in) {ind = in;)
                    public void run(int ind) {
                        long p = sum(a);
                        long q = sum2(b);
                        long r = sum3(a);
                     } // end run
        } // end indre klasse Arbeider
} // end class Problem
```

Kan vi stoppe en annen tråd ? Hvis ikke vi, hvem da?

- Helst ingen, forbudt
  - I java 1.0 var det riktignok metoder som:
    - stop() **Deprecated**. This method is inherently unsafe. Stopping a thread with Thread.stop causes it to unlock all of the monitors that it has locked ….
    - suspend() **Deprecated**. This method has been deprecated, as it is inherently deadlock-prone.
  - Disse vil bli fjernet og skaper bare problemer – **ikke** bruk dem!
- Bare trådene kan stoppe seg selv
  - midlertidig ved synkronisering
  - ved å gå ut av siste setning i sin main() eller run() metode
- En tråd kan starte andre tråder, men de må stoppe/terminere seg selv

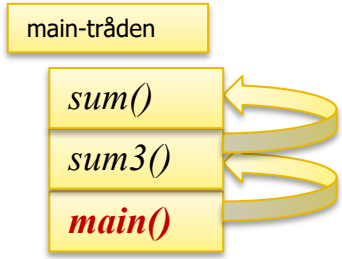
Hva er en stack (stabel)? Hvor mange har vi og hvilke er det?

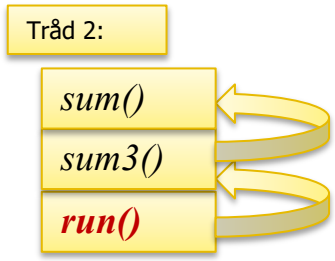
- En stabel er et dataområde som holder de metodene (med deres lokale variable og parametre) som er kalt.
  - En metode kan kalle en annen metode,
    - Da legger det nye metodeobjektet seg oppå det som kalte,..osv
    - Det hele er som en tallerken-stabel
  - Det er bare den metoden som er på toppen av en stabel som gjør noe nå.
    - De lenger nede venter bare på at den som ligger rett over den skal returnere,...osv
- Hvert trådobjekt (også objektet med main) har hver sin stabel.
  - nederst i hver av disse stablene ligger hhv. main() og run()
- Java er et multi-stabel språk (ikke alle språk er det)
- Stakkene ender seg hele tiden etter som metoder kalles og returnerer



Kan flere tråder kalle samme metode samtidig. Hva skjer da ?

- Anta at både main-tråden og tråd 2 kalle metoden sum3() som igjen kaller sum().
- Da legger det på hver av stablene en metode-objekt av hhv. sum3() og oppå det hvert sitt sum()-objekt
- **Svaret er JA**. Samtidige metodekall til samme metode skaper ingen problemer.
- Metodene 'bare ligger der' og kan kalles av alle andre metoder som har utsyn til dem, enten:
  - Gjennom sitt skop på kallstedet
  - Eller via pekere





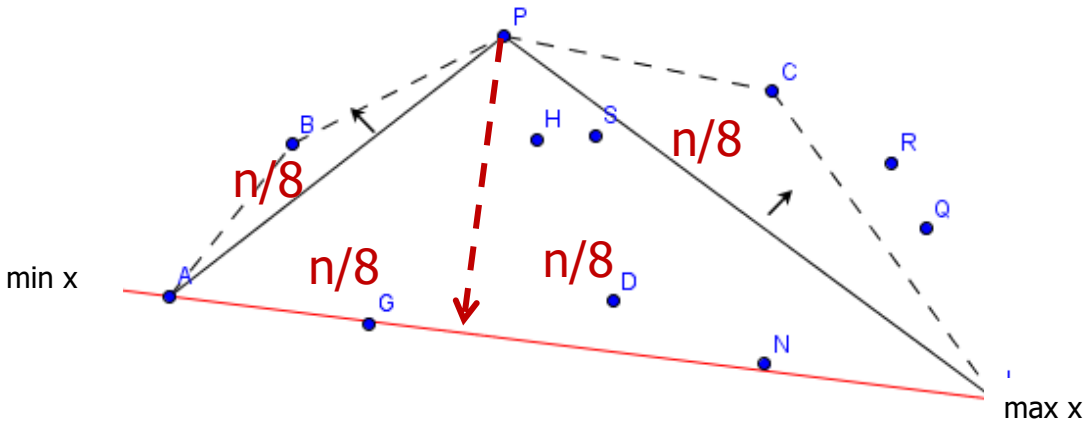
Oblig 5 – den konvekse innhyllinga (DKI) tips:

- Litt repetisjon av den sekvensielle algoritmen
  - Problemet er at vi må finn flere parametere før vi kan starte den sekvensielle, rekursive løsningen.
- Forventet kjøretid og 'verste-tilfelle' kjøretid for n punkter
- Hvilken speedup kan vi forvente
  - 2015
  - 2016
  - 2017 ?
- Hvordan løser vi sekvensielt og parallelliserer vi Oblig 5

Algoritmen for å finne den konvekse innhyllinga sekvensielt

1. Trekk linja mellom de to punktene vi vet er på innhyllinga fra maxx -minx (I - A).
2. Finn punktet med størst negativ (kan være 0) avstand fra linja (i fig 4 er det P). Flere punkter samme avstand, velg vi bare ett av dem.
3. Trekk linjene fra p1 og p2 til dette nye punktet p3 på innhyllinga (neste lysark: I-P og P-A).
4. Fortsett rekursivt fra de to nye linjene og for hver av disse finn nytt punkt på innhyllinga i størst negativ avstand (≤ 0).
5. Gjenta pkt. 3 og 4 til det ikke er flere punkter på utsida av disse linjene.
6. Gjenta steg 2-5 for linja minx-maxx (A-I) og finn alle punkter på innhyllinga under denne.

Forventet kjøretid med n tilfeldige punkter = O(n)

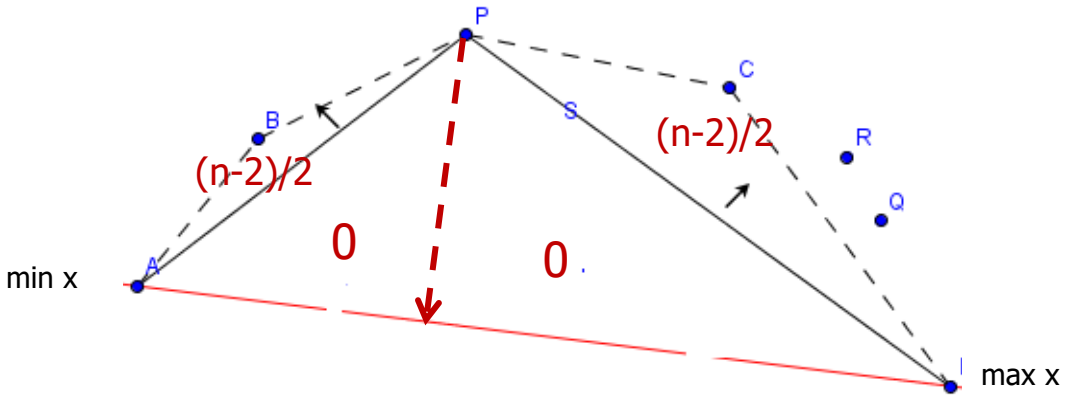


Ny mengde:

1. Finne max og min x – lete-mengde : **n**
2. Finne mengden over/under min-max og P : **n** n/2
3. Finne 2 punkter: C og B : n/2 *2 = **n**, **n/8**
4. Finne 4 punkter max-C,C-P,P-B og B-min :n/8 *4: **n/2** n/32
5. Finne 8 nye punkter : n/32 *8 : **n/4** n/128
6. Finne 16 nye punkter n/ 128 *16: **n/8** n /512
7. + ...

Sum start = **n** + Alle 'over' max-min= 2 n + n/2 + n/4 +n/8+.. = **3n**
+ sum alle 'under' min-max = **3 n**, Totalsum = n + 2*3n= **7 n = O(n)**

Verste tilfellet kjøretid – alle punktene er på innhyllinga, f.eks alle punktene på et kvadrat eller en sirkel



Ny mengde:

1. Finne max x og min x – lete-mengde : **n**
2. Finne mengden over/under min-max og P : **n** (n-2)
3. Finne 2 punkter: C og B : (n-2)/2 *2 = **n-2**, (**n-4**)
4. Finne 4 punkter :(n-4)/4 *4: **n-4** (n-8)
5. Finne 8 nye punkter (n-2^3): (n-2^4)
6. Dette fortsetter til n - $2^k \leq 1$, dvs. k= $\log_2 n$ ganger

Totalsum =

$\sum_{i=1}^{\log n}(n - 2^i) = \sum_{i=1}^{\log n} n - \sum_{i=1}^{\log n} 2^i$ = nlog n – n ≤ n $\log_2 n$ = **O(n $\log_2 n$)**

Generelt om parallellisering

A. Starten tar først **n** steg for å finne minx &maxxx
B. Så trenger vi nye **n** steg for å finne mengdene til høyre og til venstre for linja minx-maxx
C. Siden hele algoritmen tar **7n** steg, må A og B parallelliseres, ellers får vi ikke god speedup (Amdahls lov)

Alle algoritmer startet i alle fall med A.

Roadmap – to mulig gode parallelliseringer

- Metode 1: Divide-and-conquer:
  - Del punktene i k = antKjerne*overbook (=2,4,8,..)
  - Opret k tråde og kjør sekvensiell algoritme for DKI på hver mengde
  - Du har da k stk. 'små' DKI-er:
    - Del de k stk i antKjerne grupper
    - Brug antKjerne tråde til at kombinere disse små DKI til 'medium' DKI
  - Kombiner de antKjerne medium DKI til en stor DKI.

- Metode 2: Parallelliser rekursivt den sekvensielle koden.
  - Brug tråde ved rekursion i «toppen» av call-graph.

Problemer dere vil møte i den rekursive, sekvensielle løsningen II (Metode 2)

- **Finne punktene på den konvekse innhyllinga i riktig rekkefølge**?
  - Oppbyg konvekse innhyllinga et punkt av gangen.

Metode 2: Speedup Oblig5 konv. innhyll – forelesers løsning fra 2015 – som parallellisering av vanlig Quicksort

```
Test av  Parallell og sekvensiell konveks innhylling av n
punkter med 8 kjerner , og 8 traader,
 Median av:3 iterasjoner
```

n	sekv.tid(ms)	para.tid(ms)	Speedup
200000000	10213.669	6074.485	1.6814
20000000	792.634	452.406	1.7520
2000000	71.935	111.246	0.6466
200000	7.648	37.369	0.2047
20000	0.999	11.629	0.0859
2000	0.115	3.201	0.0360
200	0.016	1.223	0.0130
20	0.002	0.266	0.0090

Kommentar: Ikke veldig speedup – 1.75 på 8 kjerner
Kan klart gjøres bedre.

Metode 2: Speedup Oblig5 konv. innhyll – forelesers løsning fra 2016 – bedre parallellisering av Konveks hull, 20-200 mill

Test av parallell og sekvensiell: Den konvekse innhyllinga av n punkter med 8 kjerner , og 16 traader, Median av:3 iterasjoner

n	sekv.tid(ms)	para.tid(ms)	Speedup	KoHyll.size()
200000000	8941.138	3099.202	2.8850	18896
20000000	772.244	348.533	2.2157	6011
2000000	69.031	36.129	1.9107	1905
200000	6.782	6.726	1.0084	592
20000	0.638	3.688	0.1730	165
2000	0.098	2.192	0.0445	68
200	0.012	1.838	0.0064	20
20	0.005	1.384	0.0035	7

Kommentar: En god del raskere, nær dobbelt så rask parallelt som 2015, men speedup kunne vært noe bedre (enn 2.88). MERK 16 tråder.

Metode 1: Speedup Oblig5 konv. innhyll – forelesers løsning fra 2017 – full parallellisering av Konveks hull, 20-200 mill, ny tallfordeling

```
Test av  parallell og sekvensiell: Den konvekse innhyllinga av n punkter
 med 8 kjerner , og 8 traader,  Median av:3 iterasjoner
```

n	sekv(ms)	p1(ms)	Spdup1	p2(ms)	Spdup2	CoHull	par2CoHyll
200000000	8940.834	6821.402	1.3107	3377.267	2.6474	16626	16626
40000000	1667.866	1330.035	1.2540	554.273	3.0091	7426	7426
8000000	302.776	256.511	1.1804	122.039	2.4810	3335	3335
1600000	61.699	49.945	1.2354	19.787	3.1181	1478	1478
320000	12.089	11.528	1.0487	4.960	2.4372	660	660
64000	2.222	3.895	0.5704	1.744	1.2741	300	300
12800	0.547	0.884	0.6191	0.501	1.0917	142	142
2560	0.108	0.659	0.1631	0.564	0.1908	70	70
512	0.032	1.169	0.0272	0.759	0.0420	31	31
102	0.007	0.603	0.0115	0.476	0.0145	13	13

Kommentar: En god del raskere, dobbelt så rask parallelt som 2015 og bedre enn 2016 og viktigst, speedup >1 på små eksempler (hvor sekvensiell kjøretid er bare 0.5 ms.) ; men speedup kunne kanskje vært noe bedre (enn 3.11). MERK 8 tråder.

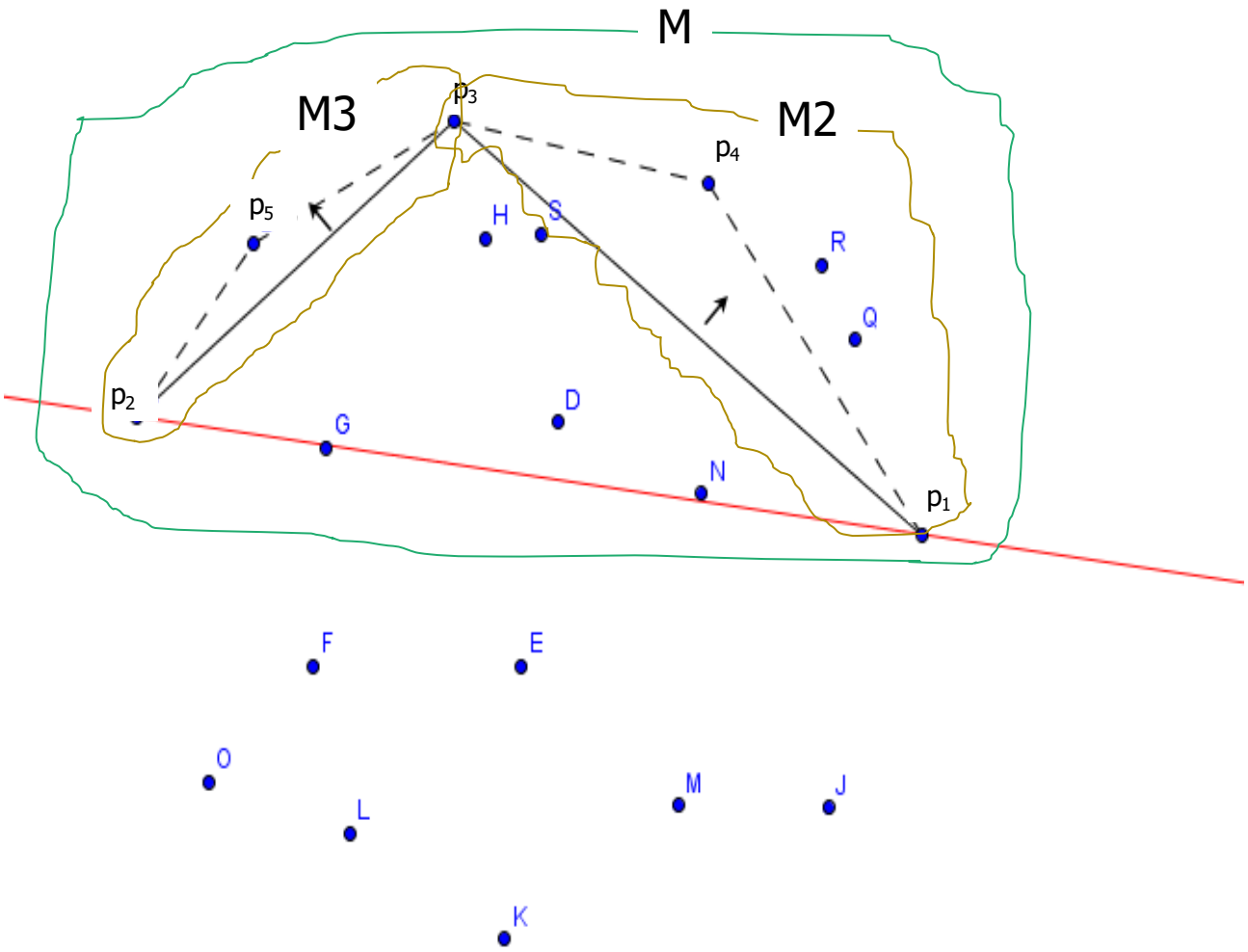
Det vanskeligste ved Oblig 5 (for meg)

- Hva skal den rekursive metoden gjøre ?
  - En idé : Ut fra:
    a) en linje p1-p2
    b) et punkt p3 som er mest negativt fra linja, og
    c) en mengde M som inneholder linja og punktet
    d) legge p2 inn i KoHyll-mengden (p1 er lagt inn) – når?
  - Skal lage:
    - Linjene: p1-p3 og p3-p2
    - Mengdene:
      - Alle punkter M1 (fra M) som er over p1-p3
      - Alle punkter M2 (fra M) som er over p3-p2
    - Nye punkter
      - Punkt P4 mest negativ avstand til p1-p3
      - Punkt p5 mest negativ avstand til p3-p2
    - (løses rekursivt; ett kall for p1-p3 og ett for p3-p2):

Rekursiv løsning: Har P_1, P_2 og P_3 (mest negativ fra P_1- P_2) og mengden M (alle over og på P_1- P_2).

Finner P_4 samtidig med M2; så P_5 samtidig med M3.

Har nå data til to nye rekursive kall.



Hvor god er denne ideen

- Fordelen med denne idéen er at vi finner mengdene og mest negativt punkt i ett gjennomløp av M
- Ulempen er at bare én av disse dataene foreligger ved start av problemet:
  - Mengden M i arrayene x[] og y[]
  - Vi mangler en linje (definert ved to punkter)
  - Vi mangler også det punktet som er lengst fra denne linja vi ikke har
- Konklusjon: Vi må finne det vi mangler først i en egen metode:
  - De to punktene: maxx og minx som definerer den første linja.
  - Mengdene M1 av punktene over eller på og M2 av punktene på eller under linja maxx-minx.

III) Parallellisering av oblig 5

Stegene i oblig 5:

- Vi jobber hele tiden med *linjer* & *mengder* (IntList) av de tallene som er kandidater til å bli valgt til nytt punkt på KoHyll og da som et endepunkt på en ny linje.
- Starten er litt vanskelig:
  1) Finn minx og maxx – trivielt å parallellisere (som oblig1)
     - Alle tallene i x[] og y[] –arrayen- Vi har da første linje.
     - Parallellisere : Trivielt jfr. FinnMax/Oblig 1
  2) Trekk linjen minx-maxx, finn nå:
     1) To mengder : Alle til venstre for (eller på) minx-maxx og alle til høyre for(eller på) minx-maxx ,
     2) Samtidig som 2.1 – finner punktet P4 = mest negativ avstand fra minx-maxx, og P3 = mest pos avstand .

  Hvordan parallellisere 2)?

  En IntList eller flere ? (en IntList for hver tråd ?)

Hva så vi på i Uke15

I) Om program, samtidige kall og synlighet av data

- Når en tråd (main eller en av de andre) aksesserer data, hvilke er det.
- Kan de parallelle trådene og main kalle samme metode samtidig. Kan de kalle metoder fra en sekvensiell løsning? Hva skjer da ?
- Hvor mange stack-er (stabler) har vi, og hvilke ?

II) Mer om et stort program, og hvordan den siste fasen i den kan parallelliseres.

- Delaunay triangulering – de beste trekantene!
  - Brukes ved kartlegging, oljeleting, bølgekraftverk,..
  - Spill-grafikk: ved å gi tekstur, farge og glatte overflater på gjenstander, personer, våpen osv.
  - Er egentlig to algoritmer etter hverandre (KoHyll + trekanttrekking)
  - Sekvensiell og parallell løsning av trekanttrekkinga.

III) Parallellisering av oblig 5 - den konvekse innhyllinga

- To strategier for parallellisering - roadmap
- Hva kan ventes av speedup på oblig 5
- De ulike stegene i oblig 5 og hvordan de alle kan parallelliseres rekursivt